Bjarne Boström

JavaFX based OPC UA Simulation Server

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 17.11.2014

Thesis supervisor:

D.Sc. Ilkka Seilonen

Thesis advisor:

M.Sc. (Tech.) Jouni Aro

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Bjarne Boström

Title: JavaFX based OPC UA Simulation Server

Date: 17.11.2014 Language: English Number of pages:7+55

Department of Automation and Systems Technology

Professorship: Information and computer systems in automation Code: AS-116

Supervisor: D.Sc. Ilkka Seilonen

Advisor: M.Sc. (Tech.) Jouni Aro

The topic of this thesis covers two relatively new technologies: OPC Unified Architecture (OPC UA) and JavaFX. OPC UA is an information modelling and exchanging standard. This thesis explains in the theoretical part the basics of OPC UA and JavaFX. The newest version of JavaFX is JavaFX 8. Because Java 8 and JavaFX 8 are bundled together, this thesis also presents some of Java 8's noteworthy new features.

The practical part of this thesis is to build an OPC UA server application called Simulation Server using JavaFX for the graphical user interface. This is done to evaluate the feasibility of using JavaFX for the graphical user interface of OPC UA and other applications implemented with Java language. The requirements for the Simulation Server are presented. Also the used tools and the finished application are shown.

The Simulation Server application is the end result of this thesis. The application can be used as a test server when developing OPC UA client applications or as a learning tool. The server provides a number of simulated signals. In addition to simulated signals there are also a number of displays for showing the internal state of the server and of the clients, which are connected to the server.

The conclusion is that JavaFX is well suited for making graphical user interfaces for Java applications. It also has features, which can be useful in non-graphical applications. It has some flaws, for example the lack of pre-built dialogs, but for most flaws and problems a future update is already scheduled.

Keywords: OPC UA, JavaFX, Java 8, Simulation Server

AALTO-YLIOPISTO
SÄHKOTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Bjarne Boström

Työn nimi: JavaFX pohjainen OPC UA -simulaatiopalvelin

Päivämäärä: 17.11.2014 Kieli: Englanti Sivumäärä:7+55

Automaatio- ja systeemitekniikan laitos

Professuuri: Automaation tietotekniikka Koodi: AS-116

Valvoja: TkT Ilkka Seilonen

Ohjaaja: DI Jouni Aro

Tämän työn aihe kattaa kaksi kohtalaisen uutta teknologiaa: OPC Unified Architecture (OPC UA) ja JavaFX. OPC UA on informaation siirtoon ja mallinnukseen tarkoitettu standardi. Tämän työn teoreettisessa osiossa käydään läpi OPC UA:n ja JavaFX:n perusteet. JavaFX:n uusin versio, JavaFX 8, on liitetty osaksi Java 8:aa, joten tässä työssä esitetään myös joitain Java 8:n huomionarvoisia uusia ominaisuuksia.

Tämän työn tavoite on toteuttaa OPC UA -sovellus "Simulation Server" (suom. Simulaatiopalvelin) käyttäen JavaFX:ää käyttöliittymän toteutukseen. Tämä tehdään siksi, että voidaan evaluoida JavaFX:n soveltuvuutta graafisten käyttöliittymien toteutukseen OPC UA ja muissakin Java-pohjaisissa sovelluksissa. Myöskin käytetyt työkalut ja valmistunut sovellus esitellään.

Simulaatiopalvelinsovellus on tämän työn lopputulos. Sovellusta voidaan käyttää apuna tehtäessä OPC UA -asiakassovelluksia tai opetustyökaluna. Palvelin tarjoaa useita simulaatiosignaaleja ja sen lisäksi näkymiä, joilla voi tarkistella sekä serverin sisäistä tilaa, että palvelimeen yhdistyneiden asiakassovellusten tilaa.

Työn tuloksena voidaan todeta, että JavaFX soveltuu hyvin Java-pohjaisten sovellusten käyttöliittymän toteutukseen. Siinä myös ominaisuuksia, joita voi hyödyntää ei-graafisella sovelluksissa. JavaFX:ssä on joitain pieniä puutteita, esimerkiksi valmiita ponnahdusikkunoita ei ole. Useimpiin puutteisiin ja ongelmiin on kuitenkin tulossa päivityksiä tulevaisuudessa.

Avainsanat: OPC UA, JavaFX, Java 8, Simulaatiopalvelin

Preface

This masters thesis was written at Prosys PMS Ltd. during the year 2014. The idea for this thesis arose when it was noticed that there is need for an OPC UA server that can simulate signals and provide comprehensive logs about its connections. I would like to thank my supervisor Ilkka Seilonen and my instructor Jouni Aro for good guidance during the project. I would also like to thank my parents, all the people working at Prosys PMS Ltd. and the people of Guild of Automation and Systems Technology for supporting me.

Otaniemi, November 17, 2014

Bjarne Boström

Contents

Abbreviations

API	Application Programming Interface
CA	Certificate Authority
CSS	Cascading Style Sheets
IDE	Integerated Development Environment
JDBC	Java Database Connectivity
JDK	Java Development Kit
JRE	Java Runtime Environment
JPA	Java Persistence API
JAXB	Java Architecture for XML Binding
OPC UA	OPC Unified Architecture
ORM	Object Relational Mapping
SDK	Software Development Kit
XML	Extensible Markup Language

1 Introduction

1.1 Background

Today communication between machines made by different vendors is becoming more and more important. In order for one entity to communicate with another a common communication method must exist. In best cases the communication protocol is a standard one, which enables multiple vendors to communicate with devices made by other vendors. OPC Unified Architecture is one of such standards. It is a secure interoperable platform independent specification.

When applications are developed they are usually tested against another simulated device. When developing client applications it is useful to have a test server to mimic the actual server or just to simulate a possible server.

Nowadays it really cannot be expected that software would run on only one operating system. Therefore when developing application they must either have one different version for each operating system or the programming language itself must be platform independent. One of these languages is Java. Java is run in a virtual machine called Java Runtime Environment (JRE). The Java Runtime Environment is available for all major operating systems (Windows, OS X and Linux).

Users expect a graphical interface. Some are able to work with command line interface only, but for the majority of people, a graphical interface is the only real usable interface. When developing Java graphical applications, until recently there has been two toolkits available, AWT and Swing. There are other toolkits available, but these two are included in the JRE. Now there is a third option: JavaFX. JavaFX seems promising technology and has received some hype during this and last year. Therefore it is useful to know whether it can be used to build real enterprise application.

1.2 Scope and objectives

The main purpose of this thesis was to design and create an OPC UA Simulation Server application. The work was done at Prosys PMS Ltd. The application would be created using two frameworks: JavaFX and Prosys OPC UA Java SDK. The following are primary reasons for developing the application:

- A test server to help customers of the SDK and others when developing OPC UA client side applications, also in workshops
- Simulating an actual production server for off-line development simulating an actual production server
- A test server to be used in interoperability test days, where it is vital to see what other clients are trying to do on the server
- A prototype for testing whether JavaFX can be used to build OPC UA based (and other) applications

- Provide marketing materials (the application itself and for the SDK)
- Provide easier testing of the server side of the SDK and help to run compliance tests
- As a by-product create reusable components for other UA/JavaFX based applications

To meet the goal of creating the Simulation Server application, the following research questions must be answered:

1. Which are the requirements for the Simulation Server application?
2. Can the Simulation Server application be developed using JavaFX?
3. How the Simulation Server application can be implemented?

The question one defines the overall requirements for the Simulation server software. In order to answer this the important features of an OPC UA server must be discovered and figure out which of those features users would want to see or log in a test server.

The second question is that can JavaFX be actually used for building the application, for example are all the required graphical components available or easily made. The standard tools required for building JavaFX applications also falls under the scope of this question.

The third question is about implementation details. One of the implementation details is figuring out how to display certain OPC UA related data. Another implementation detail is how the simulation signals should be implemented. Also under the scope of this question is which parts of the application can be made using existing components and which during development created components can be reused in future applications.

1.3 Research methods

The main research methods are literature and exploratory research by prototyping, meaning programming and testing new versions of the Simulation Server application to meet the requirements, which not all were known at the start of this thesis.

Because of the first research question OPC UA must be studied. About OPC UA there are some materials, but only one good book: the "OPC Unified Architecture" by Mahnke et al.[1]. In addition the 13-part OPC UA Specification can be used as source material. Some existing OPC UA Applications can be studied.

Because of the second question JavaFX must be researched. Literature about JavaFX (as of version 8) is sparse since the topic is quite new, some of the reference books were published during the writing of this thesis [2, 3].

1.4 Structure of the work

This thesis is organized into multiple sections. Section 2 gives background information about OPC UA and its terminology. Section 3 introduces JavaFX and because the JavaFX version used is JavaFX 8, which is bundled together with Java 8, some of Java 8's new features are also discussed as they provide substantial help when writing JavaFX code. Section 4 gives the requirements for the Simulation Server Application. Section 5 describes the tools and external libraries used for building the application. Section 6 shows the finished application. Section 7 gives conclusion and summarises the research questions, also the future improvements for the application are listed.

2 Background - OPC UA

This section gives a brief introduction to OPC Unified Architecture (OPC UA or just UA). The Address Space model of OPC UA is explained and commonly used OPC UA Services are described. This section also covers some practical aspects of developing OPC UA applications.

2.1 Definition

OPC UA is a Server Oriented Architecture (SOA) which uses a service based client-server communication using requests and responses. The OPC UA 13-part specification defines a set of services which the OPC UA server offers and client consumes. The specification is maintained by the OPC Foundation. The specification defines an extendable information model which can be extended to make more specific information modes. [1]

OPC UA has its roots in "Classic" OPC which relied on Microsoft DCOM technology [1]. The comparison of OPC UA to Classic OPC is outside of the scope of this thesis. No understanding of Classic OPC is needed for understanding OPC UA. For more information see for example the "OPC Unified Architecture" by Manhke et al.

The OPC UA specification parts are listed in Table 1. Each part of the specification is important in the context of this thesis in a sense as ultimately the Simulation Server application should be able to simulate every aspect about OPC UA. However building such an application, which would be able to simulate everything is outside the scope of this thesis. Therefore the parts 2, 3 and 4 are the most important parts. The part 2 defines security model of OPC UA. The part 3 defines how information is modelled using Nodes. The part 4 defines every service interface to OPC UA. Parts 9 and 11 could be useful in case history or alarms and events would be simulated.

Table 1: The 13 parts of OPC UA Specification [1]

Part	**Description**
Part 1: Concepts	Introduction
Part 2: Security Model	Requirements for security
Part 3: Address Space Model	Type and instance system
Part 4: Services	Service definitions
Part 5: Information Model	Standard model definition
Part 6: Service Mappings	Low level mappings of Services
Part 7: Profiles	Define useful subsets of UA functionality
Part 8: Data Access	Describing analog and digital signals
Part 9: Alarms and Conditions	Alarm types
Part 10: Programs	Defines state machines and similar types
Part 11: Historical Access	History services for data and events
Part 12: Discovery	Describes ways for clients to discovery UA servers
Part 13: Aggregates	For aggregating data using relay servers

2.2 Address Space model

The information model in OPC UA is an Address Space, which contains Nodes. Each Node has References to other Nodes. Each Node has a number of Attributes which define the Node. The Attributes are defined in the OPC UA Specification and cannot be extended. However each node can have more information by having one or more Property nodes with HasProperty Reference. Some of the Attributes are common for all nodes while others are specific for certain types of nodes. Common Attributes are listed in Table 2. The WriteMask and UserWriteMask are Optional, meaning a node can have them but they are not required.[1]

Table 2: Common Attributes which every OPC UA node has. [1]

Attribute	**Description**
NodeId	Unique identifier for the Node
NodeClass	Defines Node usage, see Table 3
BrowseName	Path name, can be used in search operations
DisplayName	User-friendly name for the node
Description	Description of the Node
WriteMask	Optional, marks which Attributes can be written
UserWriteMask	Optional, same as WriteMask, but for the current user

The nodes have different types defining their usage. The type of a node is defined by the NodeClass Attribute. The different NodeClass values are shown in Table 3. The semantics of node types in OPC UA can be compared to object-oriented progamming languages. ObjectType defines which kind of Objects instances and VariableType defines which kind of Variables where can be in the Address Space. DataTypes defines the type of the Value Attribute in Variable nodes. Objects can have Methods and they define similar semantics as methods or functions in an object oriented programming language. ReferenceTypes define the link types between nodes. Views enable creating different sub Address Spaces of the server's address space.[1]

Table 3: Different NodeClasses and their meaning[1]

NodeClass	**Description**
ObjectType	Defines types for Objects
Object	Actual instances of ObjectTypes
Method	Defines functions for Objects
VariableType	Defines types for Variables
Variable	Actual instances of VariableTypes
DataType	Defines data types for Variable values
ReferenceType	Defines the References which link nodes to each other
View	Provides special views to the address space

Nodes can be Browsed for it's references to other Nodes. References can be either Hierarchical or NonHierarchical. The hierarchical references can be thought as a

tree-like structure in the address space, and usually the address space is presented as tree-view in OPC UA Client applications [1, 4, 5].

Part of the Address Space is defined by the OPC UA Specification [1]. An example of an address space of an OPC UA server is shown in Figure 1. There is always a Root Node (which is usually not shown) and it always has the 3 child Nodes: Objects, Types and Views. The Objects node is the root for all non-type related Nodes and the Types node is the root for all type information nodes. If the Address Space of the server is large, Views can be used to reduce the number of visible nodes and show only nodes related to the the view under it. Another use-case for Views is restricting the access to only specific Nodes.

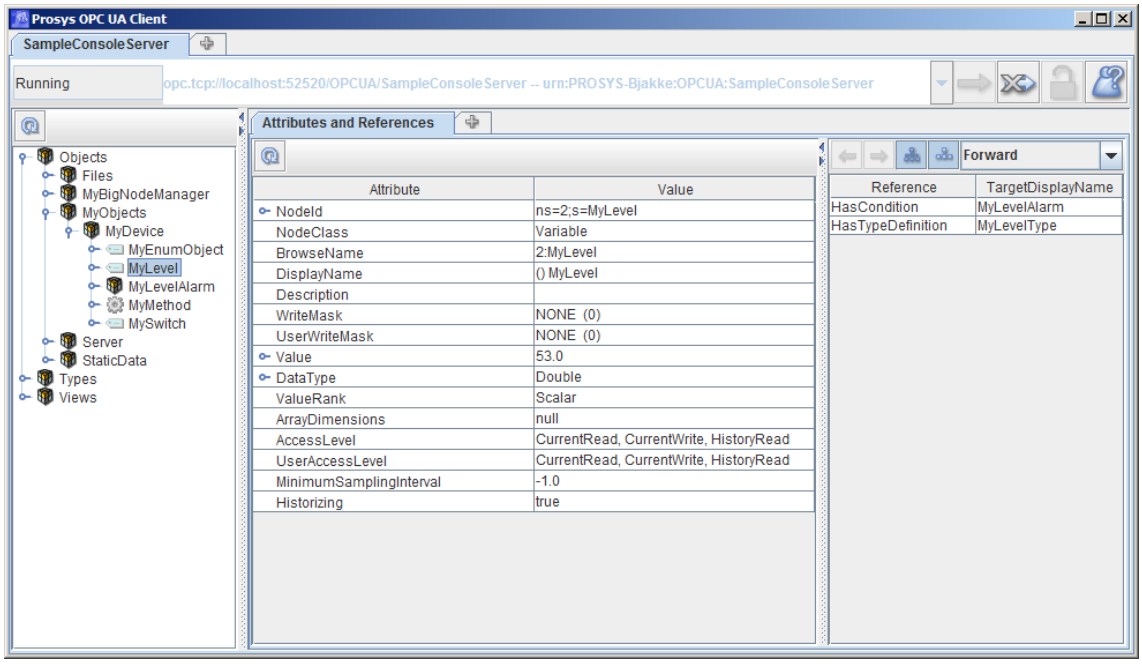

Figure 1: Prosys OPC UA Client[5]. The Address Space of the server is shown as a tree structure. The Attributes and References of selected Node are shown.

2.3 OPC UA Services

Communication in OPC UA is done using services [1]. Client sends a request to the server and the server responds. Part 4 of the OPC UA specification defines every service. The services are grouped in Services Sets. The commonly used service sets are defined in Table 4. There are also a set of profiles. If a server has a profile, then it must support the services which are defined in that profile. This enables to have a variety of devices and UA servers. For example an embedded device might not need to support all the security features of OPC UA.

The OPC UA specification part 4 defines a request and a response to each service on parameter level [1]. There are some common parameters in addition to service specific ones. For example all requests have an id and the same id is in the response for that request. Each response has a service result, which indicates whether the

Table 4: Commonly used Services of an OPC UA Server [1]

Service	**Use**
Discovery Services Set	Finding servers
Secure Channel Service Set	Form secure communication
Session Service Set	Session management
View Service Set	Browsing the Address Space
Read and Write Service	Reading and writing Attributes
Subscription Service Set	Subscribe to changes
Monitor Item Service Set	Subscribe to changes
Call Service	Call Methods
History Services	Read history data
Node Management Service Set	Adding/Removing nodes from client side

request was successful or not. Each request also has a time-out configured on the client side. In case the server does not respond in given time or the request has invalid parameters the request fails and the service result is bad. In addition to service result there are also operation results. These indicate whether a part of a request was successful. For example, the Read request can request multiple Attributes from multiple Nodes. Maybe not all reads are successful, therefore each individual read operation returns a operational status code. This enables having partial results in cases where the whole request cannot be fulfilled.

2.4 Security in OPC UA

OPC UA is designed to be secure from the start [1]. The client and the server negotiates a secure channel and after that all the communication is done using that channel. The forming of the channel is based on asymmetric public key cryptography. The exact technical implementation details are not in the scope of this thesis, but the main principle is that both the client and the server must trust each other's certificate. Both parties exchange messages and sign and encrypt using other's public key. After the initial exchange a symmetric key is negotiated, after that the symmetric key is used in encrypting messages because asymmetric crypto-operations are processor intensive tasks.

OPC UA also has options for only signing the messages without encrypting them and also the signing can be turned off to have a None security mode [1]. This can be used for example in embedded devices where the processing power is not enough for encryption support or in case the server's purpose is as such where no encryption is needed or it wouldn't be practical, for example a public test server.

Since the secure communication between the client and the server needs both to trust each other's certificates the trust decision must be done by some out-of-band mechanism, for example by hand, or by having the server's certificate signed by a Certificate Authority (CA) certificate [1]. Also it should be noted here that secure communications require that both the client and the server machine have their clock's somewhat synchronized [6].

2.5 Developing OPC UA Applications

The different software levels in OPC UA can be seen in Figure 2. The first level is the Stack. The Stacks are maintained by the OPC Foundation to ensure good interoperability. The second level is the Software Development Kit (SDK) level. The third level is the actual OPC UA Applications which are built using the SDKs. [1]

Usually a Stack provides only the bare minimum for OPC UA communication. For example the Java Stack provides ways to form the secure channel and send requests and receive responses. However, the requests and responses are plain objects which only have ways of getting and setting the value of request/responses parameters. The SDKs are built on top of the Stacks and provide better abstraction layers by for example eliminating the need to call PublishRequests manually to the server in order to receive value change for subscribed nodes. Instead the changes are received by a listener mechanism.

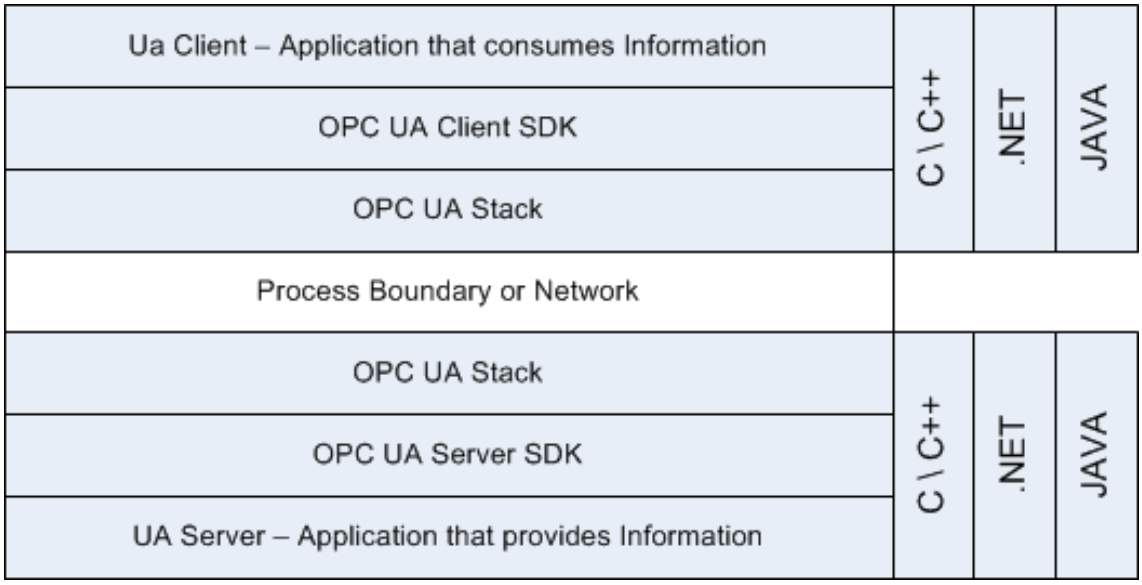

Figure 2: OPC UA software layers [1]

3 Background - JavaFX

JavaFX is the new modern Graphical User Interface (GUI) toolkit for Java. JavaFX started as an own scripting programming language called JavaFX Script and was introduced by Sun at JavaOne conference in 2007. In 2011 the next version called JavaFX 2 was released by Oracle which had aquired Sun. This time the toolkit was based on pure Java Application Programming Interface (API). As of Java 6 the JavaFX had to be separately downloaded and was not included in the Java Development Kit (JDK). In Java 7 the JavaFX and Java were bundled together, but JavaFX was not on the default classpath. [3]

Work on this thesis started when the newest JavaFX version was 2 and Java was at version 7. In 18th March 2014 Oracle released at the same time version 8 of both Java and JavaFX. This time the JavaFX was on the default classpath of Java. After the release of JavaFX 8 it was noted to be superior than JavaFX 2 and therefore should be part of this thesis. Since JavaFX 8 has a dependency on Java 8, Java 8 is also a relevant part of this thesis. Java 8 also has new features which are relevant in building graphical applications. The notation Java(FX)8 is used in this thesis to mean Java 8 + JavaFX 8.

This section introduces JavaFX, explains some differences of JavaFX 2 and 8 and also introduces some of Java 8's new features. This section only focuses on language features, it does not touch the tools which can be used to build JavaFX applications. These are explained in section 5.

3.1 Java 8

Java 8 is the newest version of Java programming language. It was released in 18th March 2014 by Oracle. It contains key improvements to the language. One of the most important new features in Java 8 is the support for lambda syntax [3].

The Lambda syntax allows sort of pass code-as-data semantics which functional programming languages usually offer. In case of Java 8 this is done by allowing the lambda syntax to implement an interface which contains exactly one method in less lines of code that an anonymous class would. This new feature is especially useful in JavaFX because many of its features rely on listeners and callback functions. With Lambda syntax a typical event handler of 5 lines becomes one-liner. An example of lambda syntax is shown in Listing 1.

Listing 1: Example of Lambda syntax advantages

```
1  package thesis.example;
2
3  import javafx.event.ActionEvent;
4  import javafx.event.EventHandler;
5  import javafx.scene.control.Button;
6
7  public class LambdaListener {
8
9          public LambdaListener() {
10                 Button button = new Button("Example");
11
12                 // Old anonymous class way of adding a listener
13                 button.setOnAction(new EventHandler<ActionEvent>() {
14                         @Override
15                         public void handle(ActionEvent event) {
16                                 doStuff();
17                         }
18                 });
19
20                 // New Lambda way of adding a listener - much shorter
21                 button.setOnAction(event -> doStuff());
22         }
23
24         private void doStuff(){
25
26         }
27 }
```

Another new feature is called Streams. Streams essentially allow efficient manipulation of data by providing a pipeline in which operations can be added. An example of using streams is shown in Listing 2.

Listing 2: Example of using Streams API

```
package thesis.example;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {

	public static void main(String[] args) {
		// First we create a number of strings of integers
		List<String> chapters = new ArrayList<String>();
		chapters.add("1");
		chapters.add("2");
		chapters.add("3");
		chapters.add("4");
		chapters.add("5");

		// Using Streams API we map them to integers
		List<Integer> numbers = chapters.stream().map(t ->
			Integer.parseInt(t)).collect(Collectors.toList());

		//we can also do calculations using the streams
		//we ca also use a method reference "Integer::parseInt" instead of "s
			-> Integer.parseInt(s)"
		double avg= chapters.stream().map(Integer::parseInt).mapToDouble(i ->
			i*2.0).average().getAsDouble();

		// Some Stream operations can be done in paraller
		long count = numbers.parallelStream().filter(t -> t > 3).count();

	}

}
```

One important new feature is the class CompletableFuture. This class enables creating actions that can be created and performed in the future. A Completable future can depend on number of other CompletableFutures. Therefore it is possible to construct event pipelines. This can be useful for example in loading an application in which parts must be constructed in specific order. An example of using CompletableFuture is shown in Listing 3.

Listing 3: Example of using CompletableFuture

```
1  package thesis.example;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.concurrent.CompletableFuture;
6  import java.util.stream.Collectors;
7
8  import javafx.application.Platform;
9
10 public class CompletableFutureExample {
11
12         public static void main(String[] args) {
13                 CompletableFuture<List<String>> cf1 =
                       CompletableFuture.supplyAsync(() -> loadEntriesTask());
14                 CompletableFuture<List<String>> cf2 = cf1.thenApplyAsync(t ->
                       filterEntriesTask(t));
15                 CompletableFuture<Void> cf3 = cf2.thenAcceptAsync(t ->
                       displayEntries(t));
16
17                 //wait for completion
18                 try {
19                         Thread.sleep(6000);
20                 } catch (InterruptedException e) {} // does not matter here
21         }
22
23         private static List<String> loadEntriesTask(){
24                 List<String> list = new ArrayList<>();
25                 // load entries from database to the list - this would take a long time
26                 try {
27                         Thread.sleep(2000);
28                 } catch (InterruptedException e) {}// does not matter here
29
30                 list.add("Entry 1");
31                 list.add("Entry 2");
32                 list.add("Entry 3");
33                 list.add("Entry 4");
34                 list.add("Entry 5");
35                 return list;
36         }
37
38         private static List<String> filterEntriesTask(List<String> entries){
39                 // perform some filtering on the entries - this would take some time
40                 // for example here we filter any entry away if it ends with "5"
41                 return entries.stream().filter(t ->
                       !t.endsWith("5")).collect(Collectors.toList());
42         }
43
44         private static void displayEntries(final List<String> items){
45                 // display items in a GUI, must be done in special JavaFX application
                       Thread
46                 Platform.runLater(() -> {
47                         // display items somewhere
48                 });
49         }
50 }
```

3.2 JavaFX Bean model and Properties

The traditional JavaBean model consists of getter and setter methods for a Property. In addition if the bean is implemented correctly, changes can be monitored using a PropertyChangeListener, which receives PropertyChangeEvents every time the property changes in the bean. Depending on how the bean is coded, it is possible to listen either a single property or all properties at once. An example of the traditional JavaBean can be found in Listing 4. Changing of different properties is differentiated by their name, meaning a constant string given by the coder of the bean. This has two drawbacks. Firstly if the name of the property changes, the constant string must be updated and secondly the programmer must remember to include the firePropertyChange line in the setter method. Also because the JavaBean model is from the era before generics in Java, all values are passed as Object in the event, therefore the events cannot be processed type safely. Use of the PropertyChangeListener API is according to Pro JavaFX 2 book "... quite cumbersome and requires quite a bit of boilerplate code" [2].

Listing 4: JavaBean example using PropertyChange API

```
1  package thesis.example;
2
3  import java.beans.PropertyChangeListener;
4  import java.beans.PropertyChangeSupport;
5
6  public class ThesisJavaBean {
7
8          private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);
9          private int pages = 42;
10
11         public void addPropertyChangeListener(PropertyChangeListener listener) {
12                 pcs.addPropertyChangeListener(listener);
13         }
14
15         public void removePropertyChangeListener(PropertyChangeListener listener) {
16                 pcs.removePropertyChangeListener(listener);
17         }
18
19         public void addPropertyChangeListener(PropertyChangeListener listener, String
               propertyName) {
20                 pcs.addPropertyChangeListener(propertyName, listener);
21         }
22
23         public void removePropertyChangeListener(PropertyChangeListener listener,
               String propertyName) {
24                 pcs.removePropertyChangeListener(propertyName, listener);
25         }
26
27         public int getPages() {
28                 return pages;
29         }
30
31         public void setPages(int pages) {
32                 int oldValue = this.pages;
33                 this.pages = pages;
34                 pcs.firePropertyChange("pages", oldValue, this.pages);
35                 this.pages = pages;
36         }
37
38         public static void main(String[] args) {
39                 ThesisJavaBean bean = new ThesisJavaBean();
40                 bean.addPropertyChangeListener(event -> {
41                         int oldValue = (int) event.getOldValue(); //unsafe cast
42                         int newValue = (int) event.getNewValue(); //unsafe cast
43                         System.out.println("old="+oldValue+", new="+newValue);
44                 }, "pages");
45
46                 bean.setPages(50); // get event old=42, new=50
47                 bean.setPages(55); // get event old=50, new=55
48         }
49 }
```

The JavaFXBean model is based on the traditional JavaBean model. It removes the concept of PropertyChangeListener and introduces a third naming convention in addition to the setter and getter in the traditional JavaBean model: property getter. Example of a JavaFXBean can be seen in Listing 5. The pages property has pagesProperty() property getter method. This property getter should return instances of Property<T> interface and since it supports Java generics, the T should be the type of the field, the only exception is that when the field is numeric data, the returned instance can also be instance of Property<Number>. The Property<T> interface supports listening value changes through ChangleListener<T>, which can be added to the Property. The Property interface has two advantages when compared to the old PropertyChangeListener API. The first is because the listener semantics are in the actual property, the programmer cannot forget adding the support it by forgetting firing the property change to the PropertyChangeSupport, which is possible in the old model and the second advantage is that the ChangeListener supports generics meaning type-safe events.

Listing 5: JavaFXBean example using Property API

```
1  package thesis.example;
2
3  import javafx.beans.property.IntegerProperty;
4  import javafx.beans.property.SimpleIntegerProperty;
5
6  public class ThesisJavaFX {
7
8          private IntegerProperty pages = new SimpleIntegerProperty(42);
9
10         public int getPages() {
11                 return pages.get();
12         }
13
14         public IntegerProperty pagesProperty(){
15                 return pages;
16         }
17
18         public void setPages(int pages) {
19                 this.pages.set(pages);;
20         }
21
22         public static void main(String[] args) {
23                 ThesisJavaFX bean = new ThesisJavaFX();
24                 bean.pagesProperty().addListener((observable, oldValue, newValue) ->
                       System.out.println("old="+oldValue+", new="+newValue));
25                 bean.setPages(50); // get event old=42, new=50
26                 bean.setPages(55); // get event old=50, new=55
27         }
28 }
```

JavaFX Property memory usage

One thing to note is memory consumption, since every property now handles it's own changes it can potentially consume much more memory than the old JavaBean PropertyChange API. Since this can affect application development a memory test was done. This test was performed on Windows 7 64-bit using Java 8u20 64-bit version from Oracle as the JRE/JDK. The memory sizes were calculated using Java VisualVM which comes with the Oracle JDK. The Old JavaBean used in the test has 10 integers and PropertyChangeSupport similar to in example in Listing 4. The JavaFX Bean has 10 IntegerProperty. Listeners are attached to each property. Exact code is presented in Appendix A. Results are shown in Table 5 and The Java VisualVM view is shown in Figure 3. The retained memory is used in the results. The retained memory is the amount of memory which would be freed if the object would be garbage collected by the JRE.

Table 5: Memory consumption test results. Old JavaBean has 10 int properties. JavaFX Bean has 10 IntegerProperties. A listener is attached to each property.

Listeners per property	**Old JavaBean (bytes)**	**JavaFX Bean (bytes)**
0	120	706
1	1658	2396
2	1738	3596
3	1818	4716

The results in Table 5 show that JavaFX beans with lots of property listeners consume much more memory than the old JavaBeans. This is something which should be kept in mind when developing JavaFX applications. However it should be noted here that there are ways to lazily load the properties which does reduce the memory consumption if the property is not listened. For example the Property could be created only if the value is changed from a default value, the standard JavaFX classes are already doing this [2].

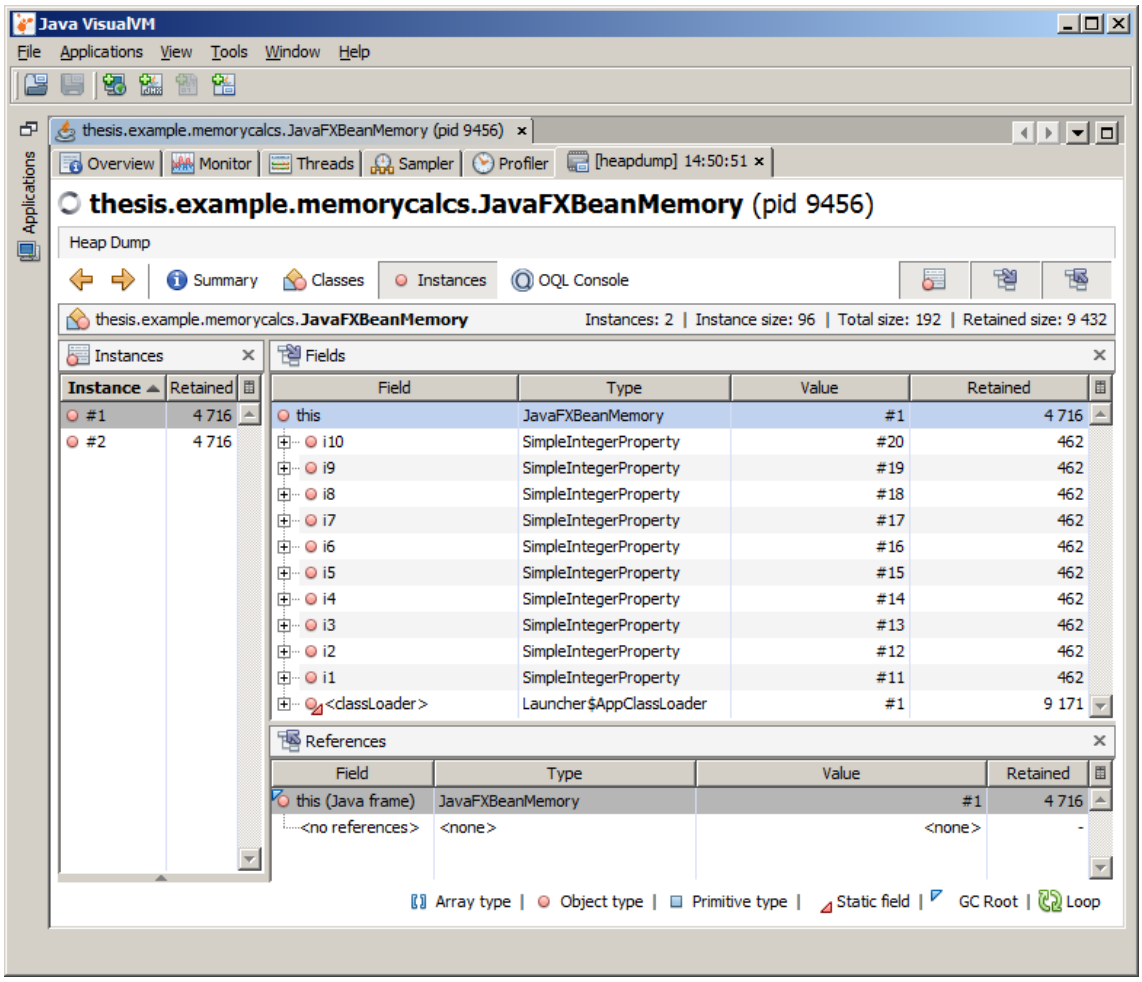

Figure 3: JavaFX Property memory usage shown in Java VisualVM application. The JavaFX Bean in question has 10 IntegerProperty and each of them has 3 ChangeListeners attached.

Bindings

The Property API also supports the concept of binding, meaning a Property can be bound to another Property either one or two-way [7]. This way the value of a Property can be linked to another Property. Bindings can also be made from any number of Observable values, which means we can make automatic calculations easily. This can be handy in layout managers, whenever for example a heightProperty changes, layout managers can resize components inside them. An example of JavaFX Bindings is shown in Listing 6.

Listing 6: Example of JavaFX Bindings

```
package thesis.example;

import javafx.beans.binding.Bindings;
import javafx.beans.binding.DoubleBinding;
import javafx.beans.property.Property;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.value.ObservableValue;

public class BindingsExample {

	public static void main(String[] args) {
		Property<String> p1 = new SimpleStringProperty("value 1");
		Property<String> p2 = new SimpleStringProperty();

		// prints "value 1 , null"
		System.out.println(p1.getValue() +" , "+p2.getValue());

		p2.bind(p1); // p2 is bound to p1

		// prints "value 1 , value 1"
		System.out.println(p1.getValue() +" , "+p2.getValue());

		p1.setValue("value 2");
		// prints "value 2 , value 2"
		System.out.println(p1.getValue() +" , "+p2.getValue());

		// a more complex binding
		Property<Number> val1 = new SimpleDoubleProperty(1.0);
		ObservableValue<Number> val2 = new SimpleDoubleProperty(1.0);
		ObservableValue<Number> val3 = new SimpleDoubleProperty(1.0);
		DoubleBinding db = Bindings.createDoubleBinding(() -> {
			double d1 = val1.getValue().doubleValue();
			double d2 = val2.getValue().doubleValue();
			double d3 = val3.getValue().doubleValue();
			return d1 + d2 + d3;
		}, val1, val2, val3);

		System.out.println(db.get()); // prints 3.0
		val1.setValue(Double.valueOf(3.0));
		System.out.println(db.get()); // prints 5.0

	}

}
```

3.3 The Scene Graph and Graphical Components

An example of a simple JavaFX application is shown in Listing 7. The Stage represents a window in the operating system. A Scene is attached to the Stage and is responsible for displaying content on the Stage window. The content is a tree-

structured hierarchy of Nodes. Every graphical component in JavaFX is a Node [7].

As of version 8, JavaFX contains a number of built-in graphical components. It has the normal set of components, which can be expected from a graphical library: Button, CheckBox, List/Table/Tree views, Text fields. It also has a number of layouts: BorderPane, AnchorPane, Hbox and Vbox. There are also other components [7]. There is one area of standard component which JavaFX lacks: dialogs. There is no support for standard dialogs like yes/no and such. It is possible of course to create your own by using multiple Stages, but they would need to be built from scratch. Dialogs will be added in a future Java(FX) version, namely 8u40 [8]. There exists also a number of third party component libraries for JavaFX. The two most important ones are JFXtras[9] and ControlsFX[10].

Listing 7: Simple JavaFX Application

```
1  package thesis.example;
2
3  import javafx.application.Application;
4  import javafx.scene.Parent;
5  import javafx.scene.Scene;
6  import javafx.scene.layout.BorderPane;
7  import javafx.stage.Stage;
8
9  public class SimpleJavaFXApplication extends Application{
10
11         public static void main(String[] args) {
12                 launch(args); //Launch the JavaFX Application
13         }
14
15         @Override
16         public void start(Stage primaryStage) throws Exception {
17                 Parent root = new BorderPane(); // Is also a JavaFX Node
18                 Scene scene = new Scene(root); // Container for Scene graph
19
20                 primaryStage.setScene(scene);
21                 primaryStage.setTitle("Simple JavaFX Application");
22                 primaryStage.show();
23         }
24
25 }
```

3.4 Threading Model

JavaFX uses a single threaded model [7]. This means every operation to live components must be done on a special JavaFX-Application Thread. Live means here a component, which is part of a Scene, which is attached to a Stage. This means that a GUI can be loaded on a separate thread but once it is visible every modification must be done on the JavaFX-Application Thread. To make this easier, JavaFX provides a way to run a Runnable in the application thread using Platform.runLater, which runs the Runnable in the application thread at some point in the future.

3.5 FXML

JavaFX has the possibility to describe the GUI layout using XML-based format called FXML. The root element of the FXML is the container, which has other nodes as it's child nodes. Listing 8 shows an example of an FXML file and Listing 9 the code to load it.

Listing 8: An example of a FXML file

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.paint.*?>

<AnchorPane id="AnchorPane" maxHeight="1.7976931348623157E308"
    maxWidth="1.7976931348623157E308" minHeight="0.0" minWidth="0.0"
    prefHeight="400.0" prefWidth="600.0" xmlns="http://javafx.com/javafx/8"
    xmlns:fx="http://javafx.com/fxml/1">
  <children>
    <Button layoutX="14.0" layoutY="14.0" mnemonicParsing="false"
        onAction="#onButtonClicked" text="Button" />
    <ListView fx:id="textArea" layoutX="14.0" layoutY="100.0" prefHeight="200.0"
        prefWidth="200.0" />
    <Label layoutX="14.0" layoutY="86.0" text="Example ListView" />
    <HBox layoutX="386.0" layoutY="14.0" prefHeight="19.0" prefWidth="200.0"
        spacing="5.0">
      <children>
        <RadioButton mnemonicParsing="false" selected="true" text="RadioButton">
          <toggleGroup>
            <ToggleGroup fx:id="testGroup" />
          </toggleGroup>
        </RadioButton>
        <RadioButton mnemonicParsing="false" selected="false" text="RadioButton"
            toggleGroup="$testGroup" />
      </children>
    </HBox>
  </children>
</AnchorPane>
```

Listing 9: Example of loading a FXML file in JavaFX

```
package thesis.example;

import java.io.IOException;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Node;
import javafx.scene.control.TextArea;

public class FXMLLoadExample {

        //This field is injected when the FXML file is loaded
        @FXML private TextArea textArea;

        public Node loadExample() throws IOException {
                FXMLLoader loader = new
                    FXMLLoader(FXMLLoadExample.class.getResource("sample.xml"));
                loader.setController(this);
                return loader.load();
        }

        //This method is automatically invoced when the button is clicked
        @FXML
        public void onButtonClicked(ActionEvent event){
                //react to button clicks
        }

}
```

3.6 CSS

The JavaFX controls are skinnable using CSS (Cascading Style Sheets) allowing much greater reuse possibilities of components [7]. These are placed in a separate .css file or can be set directly to controls by calling the setStyle method. The separation of styles from program code into a separate file enables rapid changes to style if needed. For example, the font of all Labels can be changed by just editing the CSS file.

As of version 8, JavaFX ships with a default CSS theme called Moderna. In JavaFX 2 the theme was different, called Caspian. Users only need to define changes to the default theme instead of providing a complete new style. For example it can be used to change all the fonts of the application. An example of a complete new style is AquaFX[11], which provides a style for making JavaFX look like native OS X applications.

3.7 Distribution and Deployment

A JavaFX application can be distributed normally as a .jar file and run from the command line using the "java -jar <jarfile path>" command. An average user cannot usually do this. Therefore JavaFX supports creating self-contained native install packages. The native install bundles have a private Java runtime bundled together with the application. This also removes the need to have Java Runtime installed on end user machines, which is beneficial, because users might not have the correct Java version installed.

Creating the native packages with JavaFX requires some third-party software and the native packages must be built on the target operating system. This means that for creating an installation exe-file the script must be run on a Windows machine and for a dmg-file on OS X. The disadvantage of native bundles is that they are much greater in size because of the bundled private Java Runtime.

4 Requirements

This section starts the practical part of this thesis by defining the requirements for the Simulation Server application, which is the main objective of this thesis. Some of the requirements were clear from the start, others came along during the development process either as ideas or actual needs for testing certain features of the Prosys OPC UA Java SDK. Section 5 presents the libraries and tools used in the project and section 6 describes the finished application and section 7 presents possible future improvements.

4.1 Definition

The Simulation Server is an OPC UA server application. The main purpose of the application would be to function as a test server when developing OPC UA client applications. It could also be used as a learning tool for OPC UA in workshops and similar events. It also functions as a way to test certain features of the Prosys OPC UA Java SDK. Because of these use-cases, the application should provide as much information about itself and the clients which are connected to it.

4.2 Views

The application should have a number of views, which the user would want to see when using the application. The following views should at least be in the finished application.

Status view The application should have a quick info panel. This panel should display a status info, i.e. did the server start successfully or was there any error. It should display current time and the connection addresses of the server. The time display is important, because secure communication channels require that all the machines have their clocks somewhat synchronized.

Server configuration view The endpoints of the server must be configurable. Because the application can be used in different system configurations and locations, the following settings should be configurable:

- supported protocols, opc.tcp and/or https
- network port numbers of the protocols
- server name
- supported security policies

Certificates view Certificates are a key feature in OPC UA, as major part of the security comes from asymmetric public-key cryptography. This means both the server and the client must trust each others certificates. The application should have the option to list which certificates are trusted/rejected and provide means to change rejected to trusted and vice-versa. By default it should mark all new certificates (obtained from connecting clients) rejected until changed by user.

Connection log view The application should show each connection event that does happen on the server, meaning it should show the creation, activation and closing of sessions. It would be nice to log every action the user takes on the server, for example all read and write operations performed by connected clients.

Active Sessions view The Active Session display should display the Sessions the UA server currently has. SessionId, SessionName and SessionTimeout along with Client and User Identity should be displayed. The nature of this information is such that it could be represented as a table, it could be simply a TableView component with columns for each data.

AddressSpace view The application should enable the user to browse the address space of the server. Users should be able to see the nodes in the address space and see the Attributes and References of individual nodes. This kind of functionality is usually in an OPC UA client applications [4, 5]. Considering the case of developing an OPC UA client application and using the Simulation Server as the test server, it is useful to have the address space view directly in the server to eliminate the need to have a second working client for browsing the server.

Usually this kind of view is implemented as a Tree-based view component for browsing the address space, a Table for the References and a TreeTable for the Attributes. A TreeTable is a combination of Tree and Table components. The first row of the Table is a Tree component meaning some of the rows can be expanded.

4.3 Simulation signals

The server should have a configurable number of simulation signals. These signals could be a number of different Variables or Objects. User should be able to configure signal parameters and plot them. The speed of the simulation should be configurable. User should be able to combine multiple signals together to form new signals. The signals could be implemented as JavaFX Properties, and bound to simulation time using the Bindings API. If possible, the signals could also provide history data for simulating OPC UA History Read services.

4.4 Deployment

The application should be deployable to target machines through an installer. If possible, the installer should be made using native bundling technology provided by JavaFX to bundle the JRE into the application. This is because it is expected

that most target machines do not have Java 8 installed. For those cases where it is installed a separate installer without the bundled JRE could be considered.

5 Libraries and Tools

When building modern applications it is always useful to use the available development tools, or at least know which tools are available for use. This section explains a number of libraries and tools used to build the Simulation Server application. For most parts the libraries and tools also apply for building a general JavaFX application.

5.1 Integrated Development Environment

When coding Java and so JavaFX, a good IDE(Integrated Development Environment) is a must because they help the development process considerably. There are 3 major IDEs from which to choose: Eclipse[12], NetBeans[13] and IntelliJ IDEA[14]. Each of them supports JavaFX and Java 8 either directly or through third party plug-ins. Eclipse was selected because it was familiar and support for Java(FX) 8 was good enough. The JavaFX support was added through e(fx)clipse[15] plug-in and Java 8 support through a feature patch [16].

5.2 Scene Builder

The SceneBuilder[17] is an application that enables visual editing of .fxml files. The application can be seen in Figure 4. The application enables quick preview of the FXML file being built. The preview behaves like a real program, for example if a tab in a TabView is clicked, it becomes visible like expected in a real program. SceneBuilder also supports loading CSS files to see directly the customization done by the CSS file.

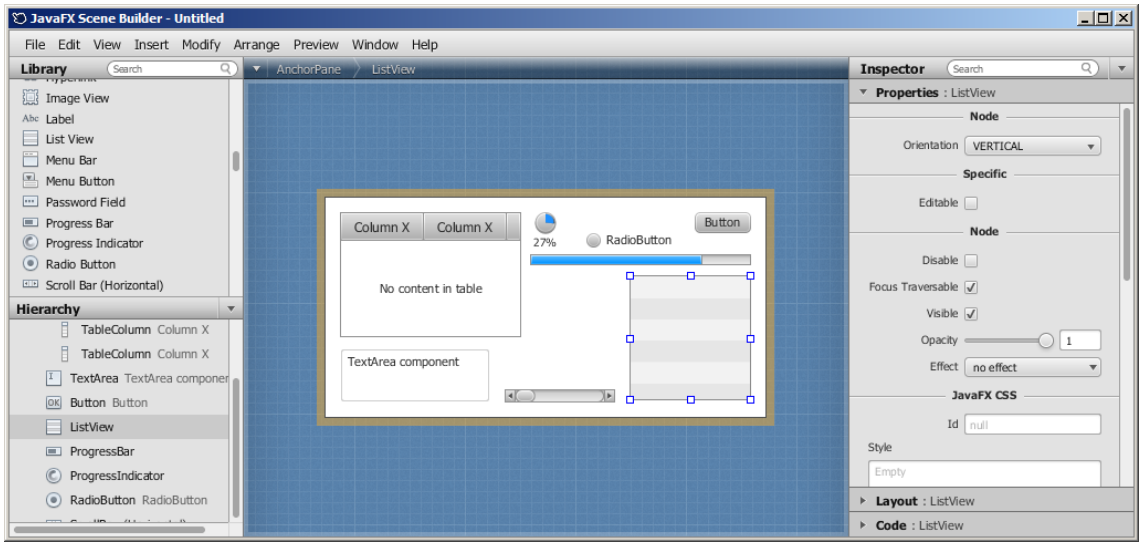

Figure 4: SceneBuilder[17] can be used to visually build FXML files. The picture shows some built-in components of JavaFX.

The ability to visually edit the graphical user interface makes development process faster, because while editing the FXML file(s) with Scenebuilder, the developer only needs to concern the visual aspect, while when editing directly the code there is a possibility to accidentally change the application logic.

5.3 Prosys OPC UA Java SDK

The OPC UA part of the application is built on top of Prosys OPC UA Java SDK provided by Prosys. Creating an OPC UA Server with the SDK is quite simple, an example is shown in Listing 10. The example creates an Uaserver instance, which is the server entry point in the program. Then the server certificates are loaded, if they exist, otherwise new certificates are created. After loading the certificates the network settings are configured. Finally the supported security modes are selected. After configuration the server is started.

When the server is created using the code shown in Listing 10 it contains only the standard information model, meaning the bare bones of the server. The server contains the type definitions and one instance of the ServerType definition, which describes the server itself. Additional information models could be loaded and/or instances created before or after starting the server.

Listing 10: Example of creating an OPC UA Server with Prosys OPC UA Java SDK

```
public class SimpleServer {

        private final UaServer server = new UaServer();
        private final String APP_NAME = "SimpleServer";

        public SimpleServer() throws Exception{
                server. setCertificateValidator (new PkiFileBasedCertificateValidator());
                ApplicationDescription appDescription = new ApplicationDescription();
                appDescription.setApplicationName(new LocalizedText(APP_NAME,
                    Locale.ENGLISH));
                appDescription.setApplicationUri("urn:localhost:OPCUA:" + APP_NAME);
                appDescription.setProductUri("urn:prosysopc.com:OPCUA:" +
                    APP_NAME);
                server.setPort(Protocol.OpcTcp, 55500);
                server.setPort(Protocol.Https, 55501);
                server.setServerName("OPCUA/"+APP_NAME);//could be set to each
                    Protocol separately
                server.setBindAddresses(EndpointUtil.getInetAddresses()); //bind to all
                    available addreses

                ApplicationIdentity identity = new ApplicationIdentity();
                identity.setApplicationDescription(appDescription);
                // <load or create certificates >
                server.setApplicationIdentity(identity);
                server.setSecurityModes(SecurityMode.ALL);
                server.getHttpsSettings(). setHttpsSecurityPolicies(
                    HttpsSecurityPolicy.ALL);
                server.addUserTokenPolicy( UserTokenPolicy.ANONYMOUS);
                server.addUserTokenPolicy(
                    UserTokenPolicy.SECURE_USERNAME_PASSWORD);
                server.addUserTokenPolicy( UserTokenPolicy.SECURE_CERTIFICATE);
                server. init ();
                server.getSessionManager().setMaxSessionCount(500); //"safety limits"
                server.getSessionManager().setMaxSessionTimeout(3600000); // one hour
                server.getSubscriptionManager().setMaxSubscriptionCount(500);
        }

        public void start() throws Exception{
                server.start();
        }

        public void stop() {
                server.shutdown(5, "Closed because...");
        }

        public static void main(String[] args) throws Exception{
                SimpleServer s = new SimpleServer();
                s.start();
                // <wait for shutdown event>
                s.stop();
        }
}
```

5.4 Database connectivity

The application will be logging lots of data. Databases are suitable for storing large amounts of data. The application will use a H2[18] database through Java Persistance API (JPA) using Hibernate ORM[19]. The H2 database is a pure Java database and has an embedded mode in where no installation is required. This makes it the perfect database for the application. The embedded mode supports both in-memory and persistence modes. Since it is useful to see connection logs even if the application is closed, the persistence mode should be used. This enables creation of a database by just specifying the connection address to a local file. The whole database is contained in the local file. JPA and Hibernate abstract the whole SQL language away which would normally be needed. Instead an entity class is annotated using Java annotations and can be stored and retrieved using a method call.

5.5 exp4j

The Simulation Signals are constructed as mathematical functions. The exp4j[20] library seems to be good in handling mathematical functions. Function can be built with any number of parameters. Later it can be evaluated multiple times by using different values for the parameters.

5.6 Jasypt

User password for user authentication testing must be stored somewhere. Since this is mainly a test software, the passwords could be stored as plain text. However since this was a good place to learn proper password storage, they are encrypted. Jasypt[21] library provides nicer API to calling encryption methods than standard Java API, therefore it will be used.

5.7 Logging

The application uses Log4j 1.2[22] as its logging library. The reason why this particular logging framework was chosen is because the Prosys OPC UA Java SDK uses this library. It is important to receive logs if the application crashes or there is any error.

5.8 Jenkins Continuous Integration Server

Jenkins[23] is a Continuous Integration Server. This means it can detect changes on version control system and trigger the building of the application. This eliminates the need to build the application manually every time a change is done. Jenkins also provides a way to run the JavaFX packaging task simultaneously on multiple computers.

5.9 General notes

JavaFX Properties can be helpful in creating linked simulation signals and for displaying changing data information. CSS enables customization of standard components for making them more suitable if some minor changes are required. However, the JavaFX Properties are much heavier in memory consumption than standard getters and setters and therefore may cause some trouble if large number of them is used.

6 Results - Simulation Server Application

This section describes the results and possible problems faced during development. The Simulation Server application and reusable components are described. The Simulation Server application is organized into a number of views. Each view is held in its own tab.

6.1 Design notes

Each of the views is modelled using Scene Builder. Scene Builder enables editing and previewing the look of the software without starting it. In the long run this saved a lot of time because starting the software takes always few minutes. The generated FXML files provide a clean separation from the look of a view and control of a view. This enables for instance changing the location of a button without affecting the result of clicking that button. Because FXML loading is used in many places in the application, a separated FXMLNode class was created to keep the loading code in one place.

6.2 Reusable components

The following reusable components were made. They are stored in a library to enable using them in different future projects.

The Connection interface The Prosys OPC UA Java SDK was lacking a good abstraction for operations which could be possible in both the client and the server. For example the Attributes of a node are known in the server, but in the client they must be read. Also in the client you can Browse the references of a Node, since server offers this service, but there is not currently a way to call the service directly from the server. Instead for most nodes you can simply ask the references from the node (since the node is data in the server).

In order to make components which could be used on both the client and the server side a common interface was needed. Therefore a Connection interface was made to encapsulate functionalities, which the components requires and two implementations of the interface, one for client and one for server side.

Address Space browsing This component provides a way to represent the Address Space in a Tree structure. It provides a selected property which can be monitored for currently selected node in the address space. Expanding a tree item is the same as asking the References of a Node, for each hierarchical reference a sub-tree node is created

Attribute view This view shows all attributes of a Node like in typical OPC UA Client applications. The view is done using a TreeTableView which has two columns: Attribute and Value. The Attribute column is a tree component, because for some Attribute values the value is structured, meaning it contains additional data.

Reference view This view shows all References of a node. User can select which References to show. User can toggle which direction Hierarchical References are shown (forward and/or inverse) and should the non-Hierarchical References be shown.

Log4j table It is important to receive logs in case there are any errors. Because of the way JavaFX Applications are deployed, there is normally no console window, which could display debugging information. Therefore it is a good idea to have a separate view, which shows every log message. The log messages could be displayed in a TableView. There is the problem of getting all the log messages. One solution is to have all loggers to log in the same place. This is not practical because each existing log statement in the SDK should be changed. Luckily the log4j framework allows creating and adding Appenders programmatically. This allows the log view component to register to receive every logging event and display them.

FXMLNode FXMLNode is a Class which extends JavaFX BorderPane. It encapsulates the code for loading a FXML file. Creating a FXMLNode requires two parameters, path where the FXML file is and the object which acts as the controller of the class.

SettingsPanel The SettingPanel enables to display arbitrary number of parameters of arbitrary classes and edit those parameters, provided that an EditorFactory is added for each class. The panel creates an editor for each parameter and displays them vertically.

PropertyPanel PropertyPanel allows displaying arbitrary number of properties of any given class in a key-value pairs. A Callback must be added for each property. The key-value pairs are displayed vertically in the panel.

PlusMinusTable PlusMinusTable is a normal table, which has small "+" and "-" labelled buttons under the table. The "+" action should be bound to an action which creates new rows to the table (or opens some editor for adding new rows of data). The "-" action deletes the currently selected row.

ValueViewer ValueViewer component allows creating a graph and add any number of UA Variable nodes to it. The graph then displays their value changes in real-time. The graph automatically scales the value axis to fit all values to screen.

6.3 Status View

The Status View, shown in Figure 5, provides a welcome screen for the user. The Server Status part changes when the application starts, after everything is started, it will display "Running". If there is any problem when starting, for example the network ports can be reserved by other applications, it will display the error message instead. The view provides two connection addresses, one for the opc.tcp protocol and one for the https protocol. It will also display the current and the starting time of the server. Displaying the time of the server is important, as secure OPC UA connections require that the client and server machines have their clocks somewhat synchronized.

Figure 5: The Status View shows current status of the application. The connection address fields can be copied for easy cut-pasting into a client application

6.4 Endpoint Settings View

The Endpoint Settings View, shown in Figure 6, allows configuring the endpoints of the server. The Simulation Server supports both the opc.tcp and the https transfer protocols, therefore both can be configured separately. The common settings for both are the network port number and the server name. Normally the server is bound to every IP address it has, but this can changed in the Bind Address part of the view.

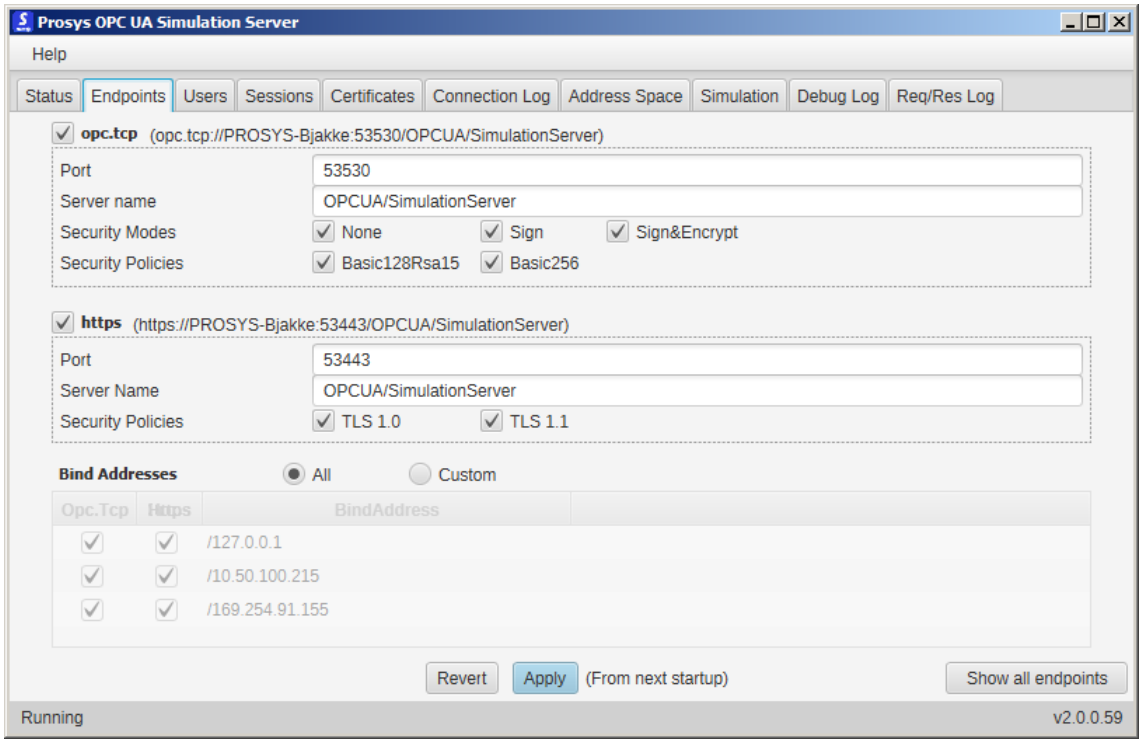

Figure 6: The Endpoint Settings View enables configuring the endpoints of the server

6.5 Users View

OPC UA has user authentication and authorization features. Access can be controlled to the node level if desired. The Users View, shown in Figure 7, allows adding and removing users for authentications. Also the passwords of the users can be changed here.

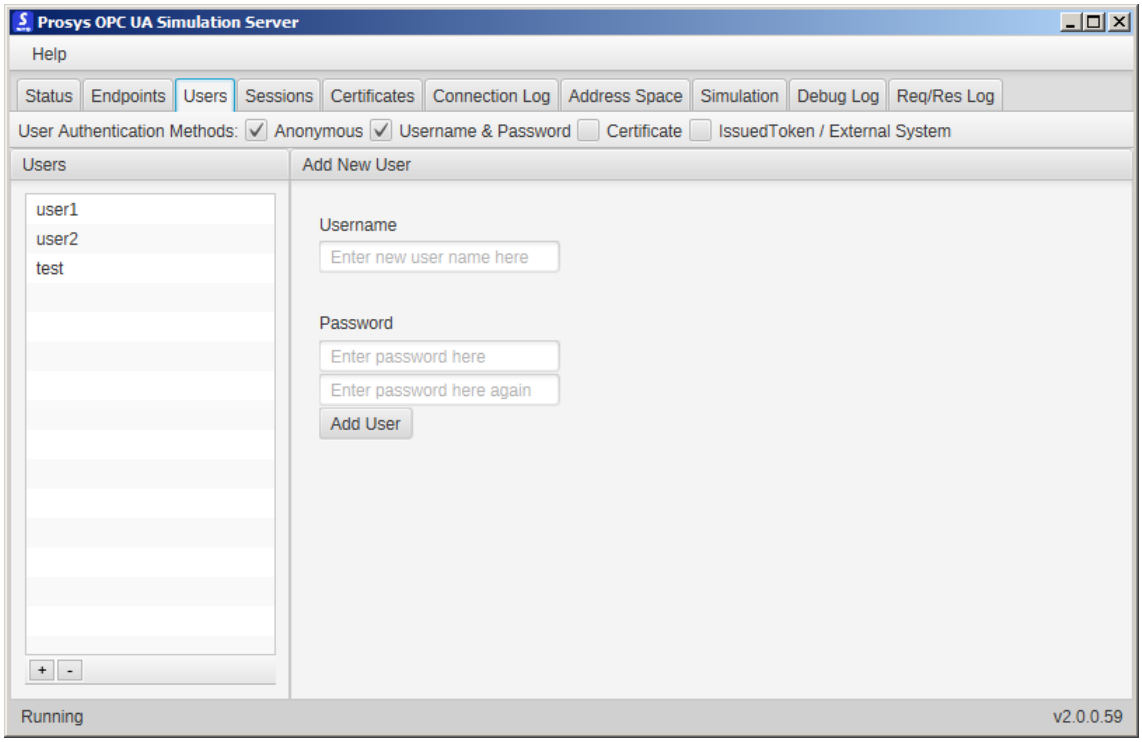

Figure 7: The Users View allows creating users to the server, which can then authenticate when connecting to the server

6.6 Sessions View

The Sessions View, seen in Figure 8, shows all currently active Sessions in the server. The list on the left shows every active session with upper row showing the SessionName of the session and lower row showing the UserIdentity of the session. By selecting a session from the list on the left, more information about the session can be seen in right side of the view.

Currently when a session is closed, it is automatically removed from the view. In the future it might be useful to show which sessions have closed instead of automatically removing them. Also there is much more Session diagnostics available from OPC UA than shown currently in the view, therefore this view could be expanded in the future.

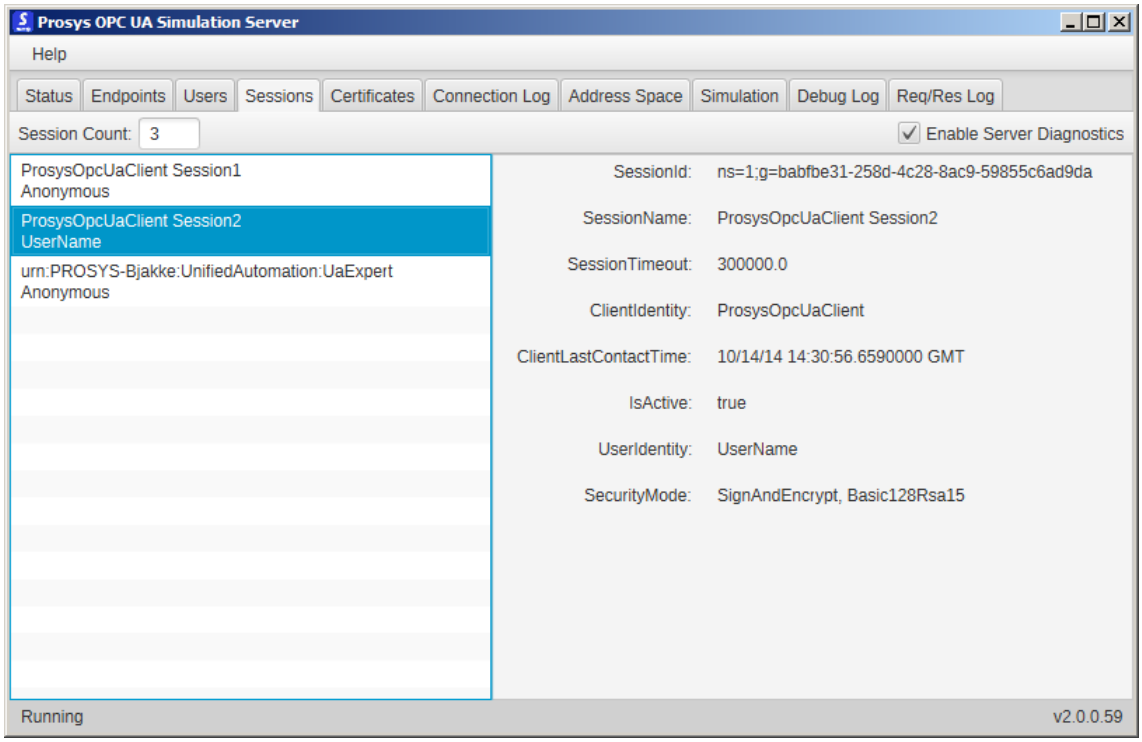

Figure 8: The Sessions View shows all currently active sessions to the server

6.7 Certificates view

Certificates are a key feature in OPC UA. They enable forming of secure communications. In order for this to work, both the client and the server must trust each other's certificate. Therefore a server application needs some way to choose which certificates are trusted. Usually the certificates are on a folder structure on a disk. Trusting a certificate normally means moving it from the rejected folder to the trusted folder. The Certificates View, seen in Figure 9, helps this by enabling users to do this through right-click menu. The view also displays key features of the certificate, e.g. when it is valid. It also contains a button "Open Certificate in OS viewer", which can be used to open the selected certificate in the default program of the operating system.

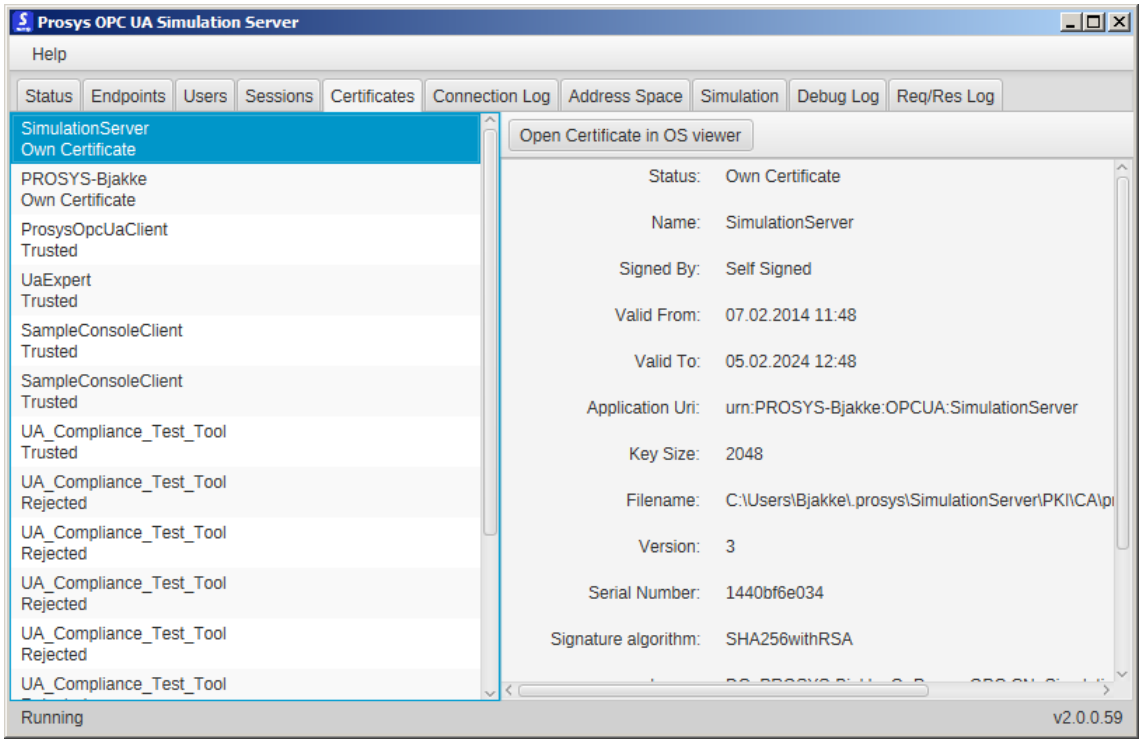

Figure 9: The Certificates View shows all certificates which the server knows. User can change rejected certificates to trusted and vise-versa.

6.8 Connection Log View

The Connection Log View, seen in Figure 10, contains a log of every each UA Session event: creation, activation and closing of the session.

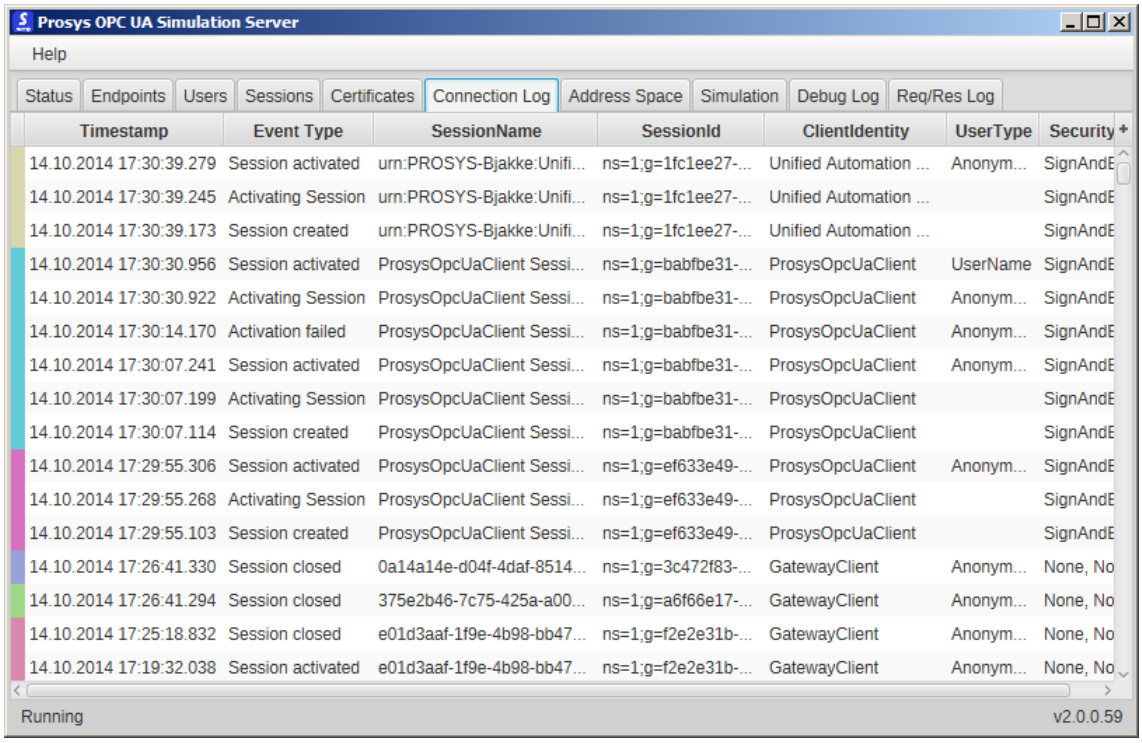

Figure 10: The Connection Log View displays every creation, activation and closing of a OPC UA Session

6.9 AddressSpace View

The AddressSpace View, shown in Figure 11, shows a view of the server's address space. This view is usually included in OPC UA client applications. It serves two purposes here. Firstly, by having this view in the server side no working client is needed when there is a need to check the server's address space. Secondly, this component is very reusable in any future client side application.

This view is usually implemented using a TreeTable component [4, 5], a table in which the first column is also a tree component. This provides a way to present short info about an Attribute on the base Attribute row and provide additional information by enabling user to expand the said row. This way a complex value can be presented clearly. This can be especially handy if an array type value must be presented.

At the start of the development, when JavaFX 2 was still the only option there was a problem: JavaFX 2 did not contain a TreeTableView component, so only one line per Attribute was possible, or multiple lines without the option to hide them. Luckily JavaFX 8 was released during the development and it contained a TreeTableView component.

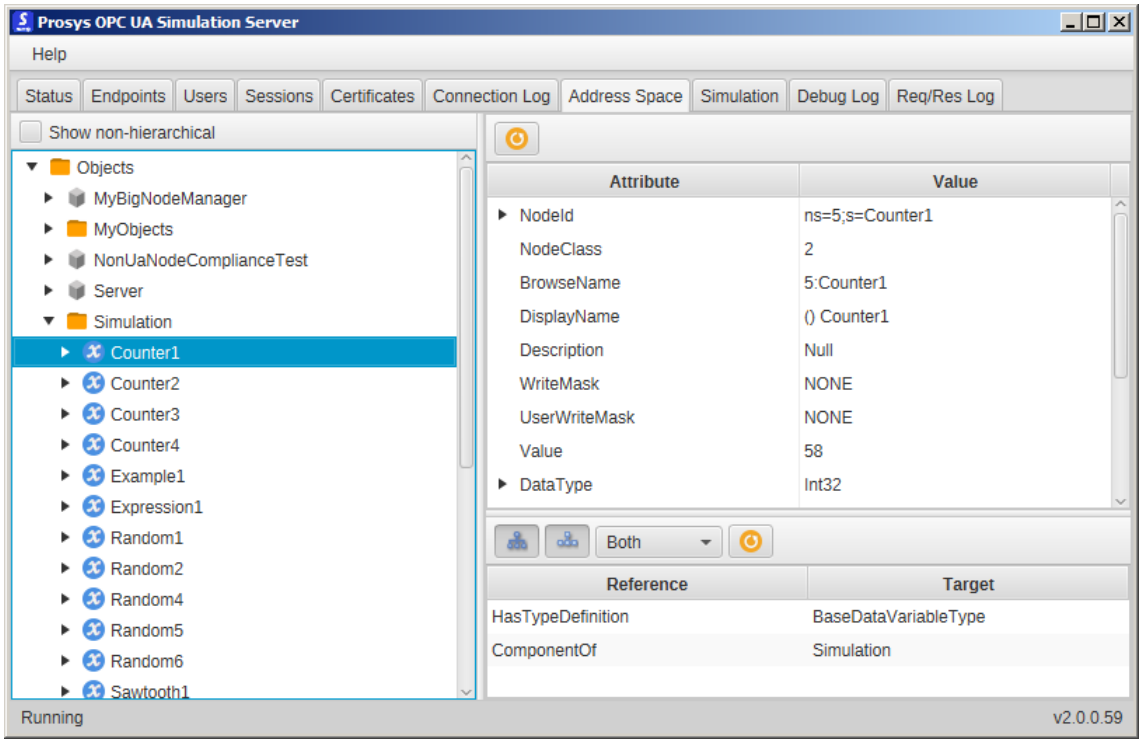

Figure 11: The Address Space View allows the user to browse the address space of the server and see Attributes and References of the selected Node

6.10 Simulation View

The Simulation View, shown in Figure 12, shows all currently simulated signals and their current values. It allows adding and removing signals, also existing signals can be reconfigured. Signals can be visualized by checking the Visualize checkbox. The user can configure how often new simulation time is calculated (milliseconds) and signal values updated. The actual signals are explained more in section 6.13.

Currently the only time source is the current time. However since the clock implementation is abstracted away from the actual signals, different clock sources could be supported in the future. This would enable for example to repeat the simulation. Currently the simulation could be repeated by changing the TimeOffset parameter of the time-based signals.

Another option for time could have been a separate simulation time, which would increase at certain intervals by a certain amount of time. However, in this case whenever the increase or interval would be changed, it could change the shape of the signal seen from the client side because the simulation time could now run faster or slower than normal time. This would mean that the signal parameters would no longer define the shape completely. This seems unwise. Therefore it is better to always have the clock run at normal speed with possible offsets.

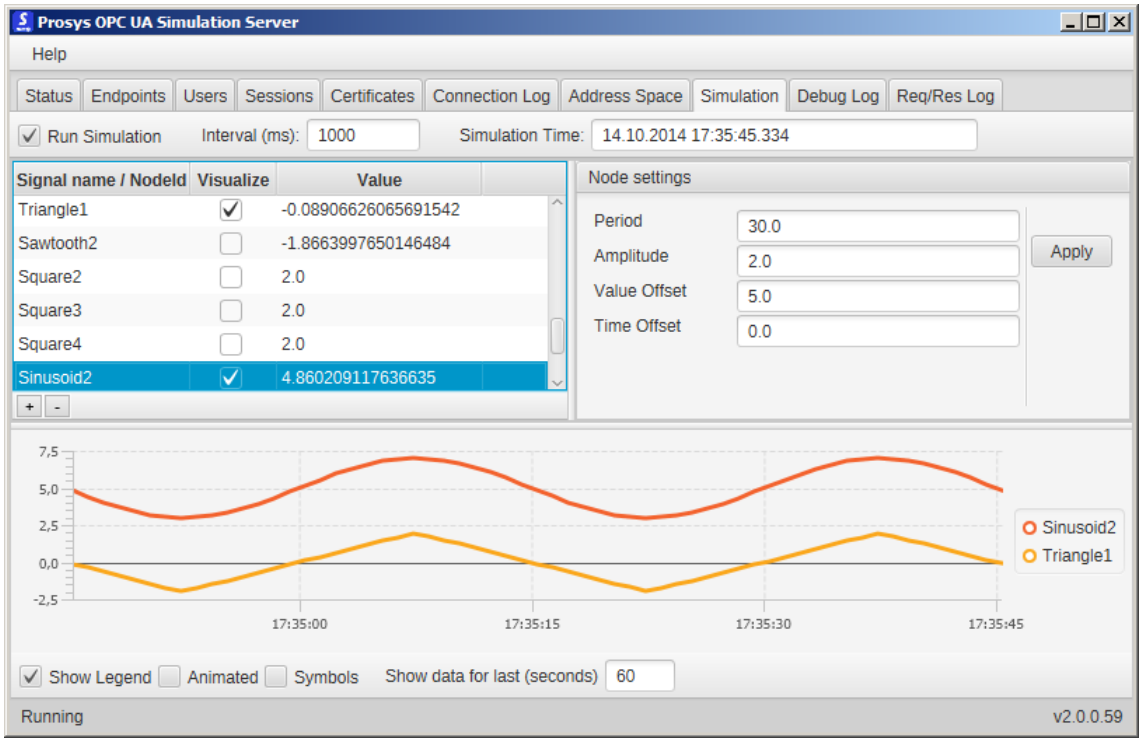

Figure 12: The Simulation View. This view displays all simulation signals and enables changing their parameters. Signals can be added, removed and visualized.

6.11 Debug Log View

The Debug Log View, seen in Figure 13, contains all debug messages usually printed on console. As in normal deployment the application does not have a console, this can be useful if debugging is needed.

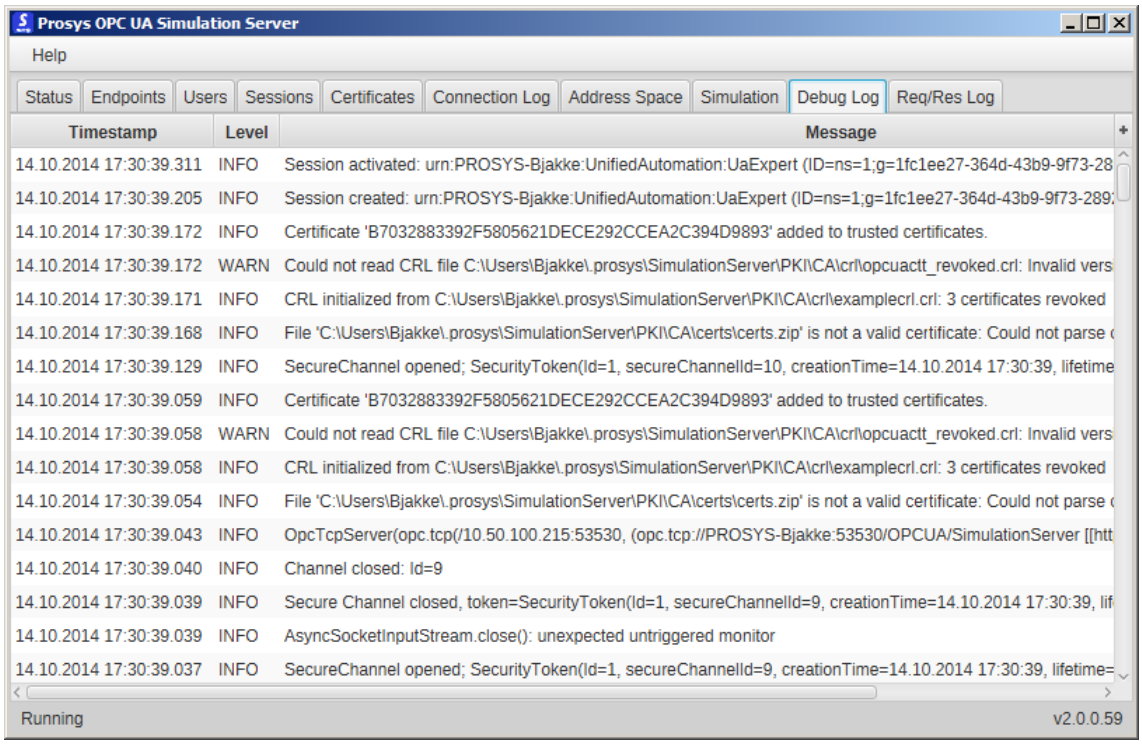

Figure 13: The Debug Log View. The View displays all logging statements, which would normally be displayed in console window.

6.12 Request-Response Log View

Since OPC UA is using Service Oriented Architecture (SOA) and uses client-server communication architecture, every service call can be determined by analysing every request made to the server and its response. Therefore it is useful for the user to be able to see them, which is what the Request-Response Log View, shown in Figure 14, allows the user to do. It enables the user to record every request-response pair which the server handles.

This view allows precise debugging of both the client application connected to the server and the actual server itself. The view is constructed in JavaFX as one ListView and two TextAreas. Buttons are placed in a Toolbar component and enable activating or deactivating the recording and clearing the existing recordings. Also the orientation of the Request and Response TextAreas can be toggled.

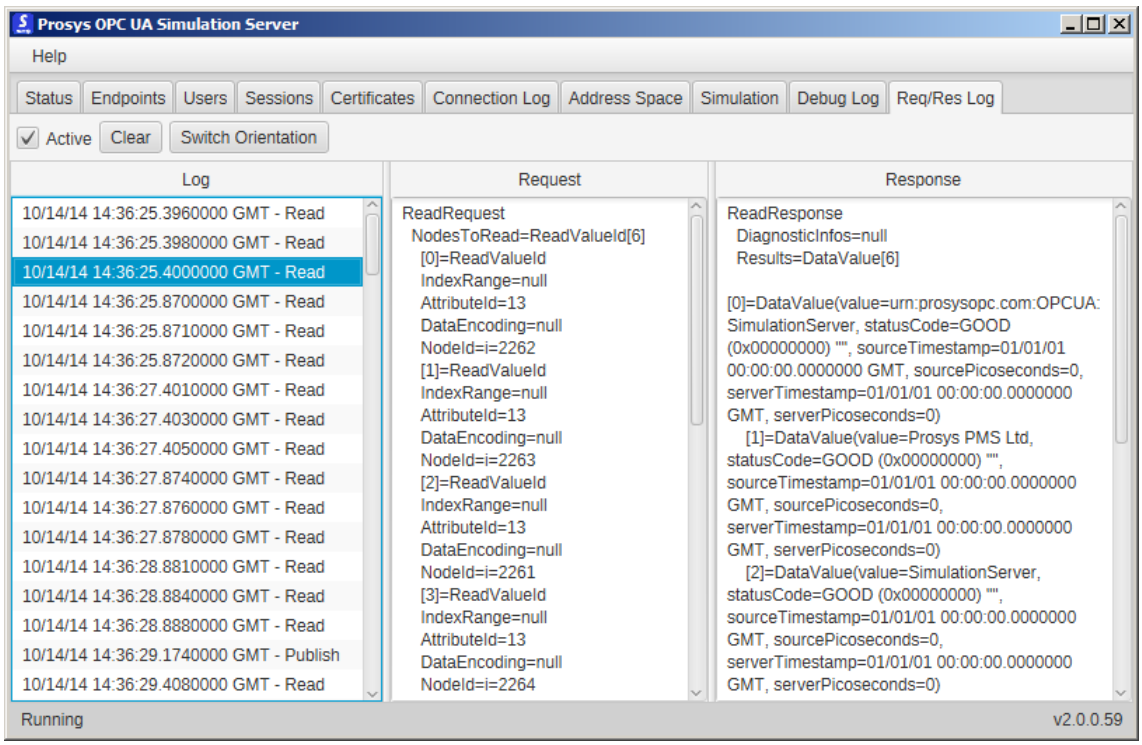

Figure 14: The Request-Respond View shows every request the server handles

6.13 Simulated Signals

Counter based signals are implemented as incrementing or decrementing their value by a certain amount at every simulation interval. Other signals are defined as mathematical functions. The simulation system provides "t" as time parameter. In addition, the signals can have configurable parameters.

Since some of the signals are functions of time, their values can be asked from whatever time. This could enable simulating history for the signals. Another way could be to store their current value in memory and return old values for their history

data. Those values would not survive restarting the application, however, if values would be written to database they could be retrieved. However, if the parameters of the signals were to change the old values would be wrong.

The signals are saved in XML format using Java Architecture for XML Binding (JAXB). JAXB enables marking Java classes with annotations and storing and loading them to and from an XML file.

Counter signal The Counter signal is a basic counter. It can count up or down or both. Multiple data types are available. The current values of counters are not saved so when server starts all counters are initialized to their configurable initial value parameter. The increment or decrement can be configured.

Random signal Random signals are implemented from Java's standard random number generator. They produce a value between 0 and 1. If other ranges are wanted, the Random signal can be combined with Expression signal to create different ranges.

Waveform signals Waveform signals are based on different variations of sine wave functions. The shape of the curve is configurable. The following signals are provided.

- Square, "a*sign(sin((2*Pi/p)*(t+to)))+vo"
- Sawtooth, "2*a*(((t+to)/p)-floor(0.5+(t+to)/p))+vo"
- Triangle, "2*a*asin(sin((2*Pi/p)*(t+to)))/Pi + vo"
- Sinusoid, "a*sin((2*Pi/p)*(t+to))+vo"

The parameters for the equations are defined below:

- a, amplitude
- p, period
- t, time
- to, time offset
- vo, value offset

Expression signal A mathematical function. Other signal types can be used as parameters in addition to simulation time. This way the signals can be combined. It is possible for example to combine a counter and a sine wave signal. The values for Expression signals are calculated after any other type of signals to prevent any other dependencies of the order of calculating signal values.

6.14 Fonts

During development it was noticed that the default font is operating system dependent and varying in length. It was also noted that even the fonts which should be available in every operating system are not always present. Therefore the only option is to embed the fonts to the application. An open-source font Liberation Sans (version 2.00) [24] was selected. The font is set using CSS and shown in Listing 11. This one example of how flexible the configuration of JavaFX is.

Listing 11: The CSS file of the Simulation Server. Includes setting fonts

```
.root{
        -fx-font-family: 'Liberation Sans';
        -fx-font-size: 9pt;

}


.table-view {
  -fx-table-cell-border-color: transparent;

}
```

6.15 Saving configuration

The configuration of the application is implemented by having a singleton class ApplicationSettings, which contains all the configurable parameters of the application. It can be accessed from every part of the software by calling ApplicationSettings.getInstance(). The singleton instance is read from and written to a config XML file by using Java Architecture for XML Binding (JAXB). The setting file is loaded when the application starts. If there are no settings file yet, a default one is created. The application saves its configuration when the application is closed (which is done by closing the main window).

6.16 Deployment

The application is deployed through platform specific installers, which bundle JRE with them. The installers are built using Jenkins continuous integration server. The builder process generates 4 installers, one for Windows platforms (.exe), one for OSX (.dmg) and two for Linux platforms (.deb and .rpm). Installers must be built on the platform it is used to install. This means that, for example, the OSX installer can only be created on the OSX operating system. Therefore building the application installer requires that the installer packaging is done once on each platform. Luckily Jenkins has the concept of nodes, meaning if the main installation runs on Windows platform, OSX and Linux machines can be added as so called slave nodes. Then we can define that a build process should be done on a certain node. The whole building job begins by the main Jenkins installation. It pulls source code from the version management and compiles the application. The process is then continued in

another job which is executed on each of the 3 nodes. The installers are then copied to the release management system.

6.17 Problems regarding deployment

There were some problems regarding the deployment. They are listed below:

- On some Microsoft Windows platforms launching the installed application would give an error about msvcr100.dll. This is a bug in JavaFX. Until the bug is fixed a workaround is to install the Microsoft Visual C++ 2010 Redistributable Package.
- The OS X version of the application is not signed by default. This means the application must be manually started the first time.
- If the application is built using 64 bit JDK, the application wont work on a 32 bit machine, because the bundled JRE is 64 bit. This problem was solved by building the application on machines using 32 bit JREs.

7 Conclusions and future work

This section ends this thesis by summarising the answers to the research question stated in section 1.2. Possible future improvements to the Simulation Server application is also discussed.

JavaFX can be used for application development in it's current state (8u20). Developing JavaFX applications on Java 6 or 7 version where only JavaFX 2 is available is not recommended because JavaFX 2 will get only security updates and not any new features and there will be major new features, for example dialogs, in the future versions.

JavaFX bean model is superior compared to old JavaBean model when observable Java properies are required. The only downside is that the JavaFX Properties can consume more memory and there is a possibility to leak memory if the listeners are not removed correctly.

Which are the requirements for the Simulation Server application?

This question was answered in section 4. The main requirement for the Simulation Server application is to provide flexible configuration of simulation signals. Additional requirements were to provide as much logs and information both from itself and from connected clients. One important feature is the ability to show the address space of the server.

Can the Simulation Server application be developed using JavaFX?

As shown in sections 3, 5 and 6, the JavaFX is a powerful graphical toolkit. It has enough built-in components so OPC UA related data can be displayed. The final required piece came with JavaFX 8's TreeTableView addition. It is the perfect component to display the structured data of OPC UA.

How the Simulation Server application can be implemented?

Answer to this question can be found in 6. It seems the best way to model signals is as function of time. This way the signal is always the same, only the sampling of it changes. This is similar to real signals. By having the signal being function of time there is the possibility to reproduce the same signal values.

The option for creating native installers makes the distribution and deployment of the application convenient. It seems the the future of Java applications is to embedded the JRE to the application.

Future improvements

The finished application could use some future improvements. Some of the most important ones are:

- Simulation of history. Since most signals are functions of time, history could be simulated by creating history values on the fly when history is requested

- Enable viewing request per session
- Would be nice if installers could be built on just one machine and not require all 3 platforms
- Support other clocks than current time
- Support external value generation for signals
- Perhaps the Sessions View should keep removed sessions or provide a configurable option
- Support user authorization simulation for node level access (only authentication supported for now)

References

[1] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3540688986, 9783540688983.

[2] J. Weaver et al. *Pro JavaFX 2: A Definitive Guide to Rich Clients with Java Technology*. Apresspod Series. Apress, 2012. ISBN: 9781430268727. URL: http://www.apress.com/9781430268727.

[3] Cay S. Horstmann. *Java SE 8 for the Really Impatient*. 1st. Addison-Wesley Professional, 2014. ISBN: 0321927761, 9780321927767.

[4] *UaExpert*. URL: http://www.unified-automation.com/products/development-tools/uaexpert.html (visited on 10/31/2014).

[5] *Prosys OPC UA Client*. URL: https://www.prosysopc.com/products/opc-ua-client/ (visited on 10/31/2014).

[6] *OPC Unified Architecture Specification, Part 6: Mappings*. Release 1.02, 2012. URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-6-mappings/ (visited on 11/16/2014).

[7] *Mastering JavaFX 8 Controls*. 1st. McGraw-Hill Education, 2014. ISBN: 9780071833776.

[8] *JavaFX Jira bugtracker, issue RT-12643*. URL: https://javafx-jira.kenai.com/browse/RT-12643 (visited on 09/18/2014).

[9] *JFXtras*. URL: http://jfxtras.org/ (visited on 09/14/2014).

[10] *ControlsFX*. URL: http://fxexperience.com/controlsfx/ (visited on 09/14/2014).

[11] *AquaFX*. URL: http://aquafx-project.com/ (visited on 10/24/2014).

[12] *Eclipse*. URL: https://www.eclipse.org/home/index.php (visited on 09/18/2014).

[13] *NetBeans IDE*. URL: https://netbeans.org/ (visited on 10/31/2014).

[14] *IntelliJ IDEA*. URL: http://www.jetbrains.com/idea/ (visited on 10/31/2014).

[15] *e(fx)clipse*. URL: http://www.eclipse.org/efxclipse/index.html (visited on 10/31/2014).

[16] *JDT/Eclipse Java 8 Support For Kepler*. URL: https://wiki.eclipse.org/JDT/Eclipse_Java_8_Support_For_Kepler (visited on 09/18/2014).

[17] *JavaFX Scene Builder*. URL: http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html (visited on 10/31/2014).

[18] *H2 Database Engine*. URL: http://www.h2database.com/html/main.html (visited on 10/12/2014).

[19] *Hibernate ORM*. URL: http://hibernate.org/orm/ (visited on 09/14/2014).

[20] *exp4j*. URL: http://www.objecthunter.net/exp4j/index.html (visited on 09/14/2014).

[21] *Jasypt: Java simplified encryption*. URL: http://www.jasypt.org/ (visited on 09/14/2014).

[22] *Apache log4j 1.2*. URL: http://logging.apache.org/log4j/1.2/ (visited on 09/18/2014).

[23] *Jenkins CI*. URL: http://jenkins-ci.org/ (visited on 09/20/2014).

[24] *liberation-fonts*. URL: https://fedorahosted.org/liberation-fonts/ (visited on 10/11/2014).

Appendix A

Listing 12: Code for measuring memory consumption of old JavaBean object when having property listeners

```
1  package thesis.example.memorycalcs;
2
3  import java.beans.PropertyChangeListener;
4  import java.beans.PropertyChangeSupport;
5  import java.io.IOException;
6
7  public class OldJavaBeanMemory {
8
9          private int i1 = 42;
10         private int i2 = 42;
11         private int i3 = 42;
12         private int i4 = 42;
13         private int i5 = 42;
14         private int i6 = 42;
15         private int i7 = 42;
16         private int i8 = 42;
17         private int i9 = 42;
18         private int i10= 42;
19
20         private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);
21
22         public OldJavaBeanMemory() {
23
24         }
25
26         /**
27          * @return the i1
28          */
29         public int getI1() {
30                 return i1;
31         }
32
33         /**
34          * @param i1 the i1 to set
35          */
36         public void setI1(int i1) {
37                 this.i1 = i1;
38         }
39
40         /**
41          * @return the i2
42          */
43         public int getI2() {
44                 return i2;
45         }
46
47         /**
48          * @param i2 the i2 to set
49          */
```

```
50          public void setI2(int i2) {
51                  this.i2 = i2;
52          }
53
54          /**
55           * @return the i3
56           */
57          public int getI3() {
58                  return i3;
59          }
60
61          /**
62           * @param i3 the i3 to set
63           */
64          public void setI3(int i3) {
65                  this.i3 = i3;
66          }
67
68          /**
69           * @return the i4
70           */
71          public int getI4() {
72                  return i4;
73          }
74
75          /**
76           * @param i4 the i4 to set
77           */
78          public void setI4(int i4) {
79                  this.i4 = i4;
80          }
81
82          /**
83           * @return the i5
84           */
85          public int getI5() {
86                  return i5;
87          }
88
89          /**
90           * @param i5 the i5 to set
91           */
92          public void setI5(int i5) {
93                  this.i5 = i5;
94          }
95
96          /**
97           * @return the i6
98           */
99          public int getI6() {
100                 return i6;
101         }
102
103         /**
```

```
104             * @param i6 the i6 to set
105             */
106            public void setI6(int i6) {
107                    this.i6 = i6;
108            }
109
110            /**
111             * @return the i7
112             */
113            public int getI7() {
114                    return i7;
115            }
116
117            /**
118             * @param i7 the i7 to set
119             */
120            public void setI7(int i7) {
121                    this.i7 = i7;
122            }
123
124            /**
125             * @return the i8
126             */
127            public int getI8() {
128                    return i8;
129            }
130
131            /**
132             * @param i8 the i8 to set
133             */
134            public void setI8(int i8) {
135                    this.i8 = i8;
136            }
137
138            /**
139             * @return the i9
140             */
141            public int getI9() {
142                    return i9;
143            }
144
145            /**
146             * @param i9 the i9 to set
147             */
148            public void setI9(int i9) {
149                    this.i9 = i9;
150            }
151
152            /**
153             * @return the i10
154             */
155            public int getI10() {
156                    return i10;
157            }
```

```
158
159         /**
160          * @param i10 the i10 to set
161          */
162         public void setI10(int i10) {
163                 this.i10 = i10;
164         }
165
166         public void addPropertyChangeListener(PropertyChangeListener listener) {
167                 pcs.addPropertyChangeListener(listener);
168         }
169
170         public void removePropertyChangeListener(PropertyChangeListener listener) {
171                 pcs.removePropertyChangeListener(listener);
172         }
173
174         public void addPropertyChangeListener(PropertyChangeListener listener, String
              propertyName) {
175                 pcs.addPropertyChangeListener(propertyName, listener);
176         }
177
178         public void removePropertyChangeListener(PropertyChangeListener listener, String
              propertyName) {
179                 pcs.removePropertyChangeListener(propertyName, listener);
180         }
181
182         public static void main(String[] args) {
183                 PropertyChangeListener listener = evt -> System.out.println(evt);
184
185                 OldJavaBeanMemory bean = new OldJavaBeanMemory();
186                 OldJavaBeanMemory bean2 = new OldJavaBeanMemory();
187                 int listeners = 3;
188
189                 for(int count = 0; count<listeners; count++){
190                         for(int i =1; i< 11; i++){
191                                 bean.addPropertyChangeListener(listener, "i"+i);
192                         }
193                 }
194
195                 for(int count = 0; count<listeners; count++){
196                         for(int i =1; i< 11; i++){
197                                 bean2.addPropertyChangeListener(listener, "i"+i);
198                         }
199                 }
200
201                 //wait until key press to enable profiling
202                 try {
203                         System.in.read();
204                 } catch (IOException e) {
205                 }
206
207         }
208
209 }
```

Listing 13: Code for measuring memory consumption of new JavaFX Bean object when having property listeners

```
1  package thesis.example.memorycalcs;
2
3  import java.io.IOException;
4
5  import javafx.beans.property.IntegerProperty;
6  import javafx.beans.property.SimpleIntegerProperty;
7  import javafx.beans.value.ChangeListener;
8
9  public class JavaFXBeanMemory {
10
11         IntegerProperty i1 = new SimpleIntegerProperty(42);
12         IntegerProperty i2 = new SimpleIntegerProperty(42);
13         IntegerProperty i3 = new SimpleIntegerProperty(42);
14         IntegerProperty i4 = new SimpleIntegerProperty(42);
15         IntegerProperty i5 = new SimpleIntegerProperty(42);
16
17         IntegerProperty i6 = new SimpleIntegerProperty(42);
18         IntegerProperty i7 = new SimpleIntegerProperty(42);
19         IntegerProperty i8 = new SimpleIntegerProperty(42);
20         IntegerProperty i9 = new SimpleIntegerProperty(42);
21         IntegerProperty i10 = new SimpleIntegerProperty(42);
22
23         public JavaFXBeanMemory() {}
24
25         public IntegerProperty i1Property(){ return i1; }
26         public IntegerProperty i2Property(){ return i2; }
27         public IntegerProperty i3Property(){ return i3; }
28         public IntegerProperty i4Property(){ return i4; }
29         public IntegerProperty i5Property(){ return i5; }
30
31         public IntegerProperty i6Property(){ return i6; }
32         public IntegerProperty i7Property(){ return i7; }
33         public IntegerProperty i8Property(){ return i8; }
34         public IntegerProperty i9Property(){ return i9; }
35         public IntegerProperty i10Property(){ return i10; }
36
37         public static void main(String[] args) {
38                 ChangeListener<Number> listener = (observable, oldValue, newValue) ->
                       System.out.println("old="+oldValue+", new="+newValue);
39
40                 JavaFXBeanMemory bean = new JavaFXBeanMemory();
41                 JavaFXBeanMemory bean2 = new JavaFXBeanMemory();
42                 int listeners = 3;
43
44                 for(int count = 0; count<listeners; count++){
45                         for(int i =1; i< 11; i++){
46                                 bean.i1Property().addListener(listener);
47                                 bean.i2Property().addListener(listener);
48                                 bean.i3Property().addListener(listener);
49                                 bean.i4Property().addListener(listener);
50                                 bean.i5Property().addListener(listener);
51                                 bean.i6Property().addListener(listener);
```

```
52                                  bean.i7Property().addListener(listener);
53                                  bean.i8Property().addListener(listener);
54                                  bean.i9Property().addListener(listener);
55                                  bean.i10Property().addListener(listener);
56                          }
57                  }
58
59                  for(int count = 0; count<listeners; count++){
60                          for(int i =1; i< 11; i++){
61                                  bean2.i1Property().addListener(listener);
62                                  bean2.i2Property().addListener(listener);
63                                  bean2.i3Property().addListener(listener);
64                                  bean2.i4Property().addListener(listener);
65                                  bean2.i5Property().addListener(listener);
66                                  bean2.i6Property().addListener(listener);
67                                  bean2.i7Property().addListener(listener);
68                                  bean2.i8Property().addListener(listener);
69                                  bean2.i9Property().addListener(listener);
70                                  bean2.i10Property().addListener(listener);
71                          }
72                  }
73
74                  //wait until key press to enable profiling
75                  try {
76                          System.in.read();
77                  } catch (IOException e) {
78                  }
79          }
80
81  }
```